MapReduce for Repy

Alper Sarikaya

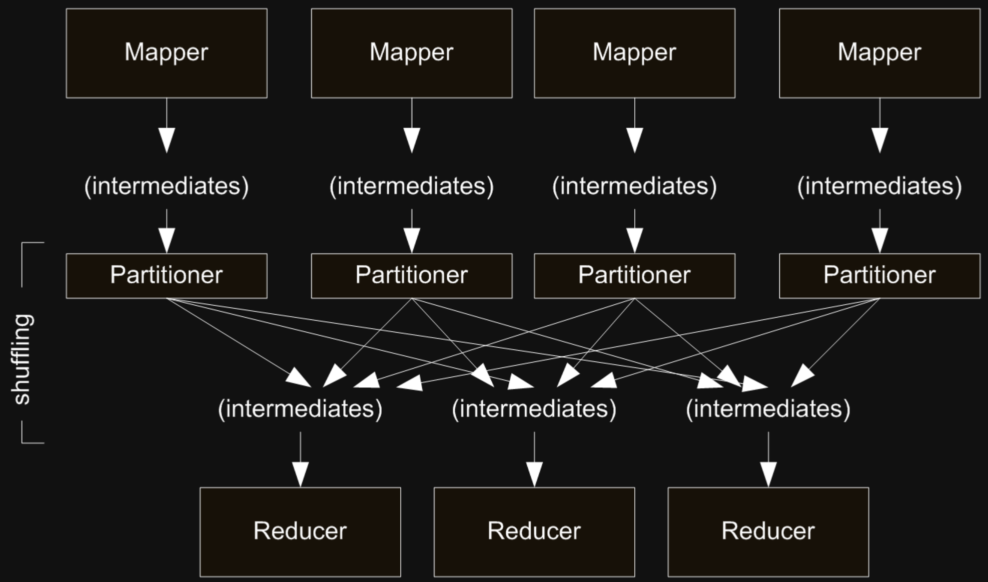
Dist. Sys. Capstone – CSE 490H

March 18, 2009

alpers@cs.washington.edu

Quick Recap of MapReduce

- Functional programming is powerful!
- Easy to parallelize map() and reduce() passes on data
- Utilizing multiple nodes, a MapReduce implement-ation must also be fault-tolerance as to not waste work
- Great for pre-computing indices and repetitive tasks

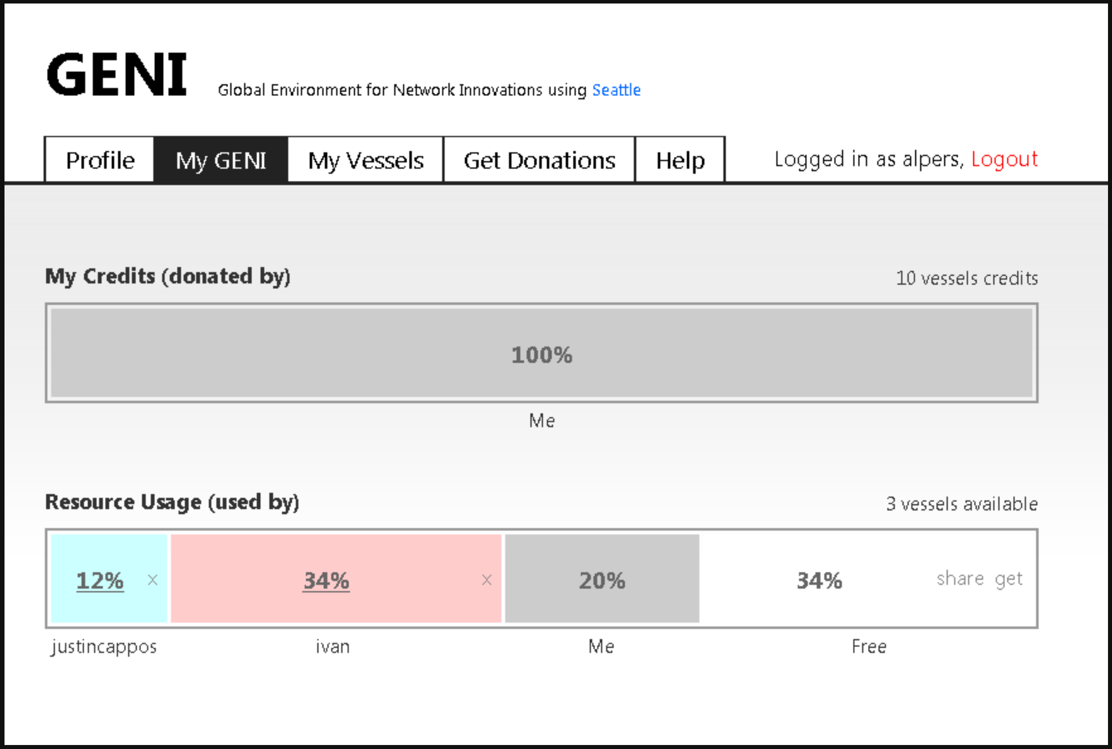


Seattle: The Internet as a Testbed

- A platform for education use for networking and distributed systems research & teaching
- Initiated by Justin Cappos, post doc. at UW
- A time-sharing application similar to SETI@home or Rosetta@home
  - Instead of running only when idle, Seattle uses up to 10% of a machine's resources (fully-customizable)
    - This include HD space, RAM usage, CPU usage, port usage/binding, thread-spawning, etc..

Seattle: The Internet as a Testbed

- Users of Seattle can acquire nodes through GENI
- Can use a shell-like interface (*seash*) to connect to vessels and run Repy code



Repy ⊂ Python

- Since vessels are not fully virtualized, need to create a safe language
  - Repy aims to be secure, robust, and simple
- Repy limits the use of hazardous calls
  - e.g. bin, callable, delattr, dir, eval, execfile, globals, input, iter, raw_input, reload, staticmethod, super, unicode, __import__
  - Cannot dynamically import code
- Repy provides nice abstracted constructs
  - e.g. sockobj.recv(52) will block until 52 b recv'd

Example of Repy Code

```
def get_data(ip, port, socketobj, thiscommhandle, listenhandle):
    # get a list of all of our neighbors!
    mycontext['primary'] = recv_message(socketobj)
    print "Primary init thread: got primary loc:", mycontext['pri']

    # we need to know how many peers we have..
    mycontext['num_peers'] = int(socketobj.recv(4))
    print "Primary init thread: got num_peers: ",
        mycontext['num_peers']

    mycontext['peers'] = []
    for i in range(mycontext['num_peers']):
        mycontext['peers'].append(recv_message(socketobj))

    # parse and save data file:
    buf = recv_message(socketobj)
    print "Primary init thread: got file data"
    dataobj = open("map_data.dat", "w")
    dataobj.write(buf)
    dataobj.close()
```

How does Repy code affect porting MapReduce functionality?

- Code to be imported (e.g. include mapper.repy) must be pre-processed by repypp.py
  - repypp.py simply copies the included file into the current file; skips include loops
    - This isn't dynamic in the least!
  - Impossible with current Seattle implementation to utilize new map(), partition(), hash(), reduce() methods on the fly
- Since python module pickle can't be used, have to make serialization from scratch!

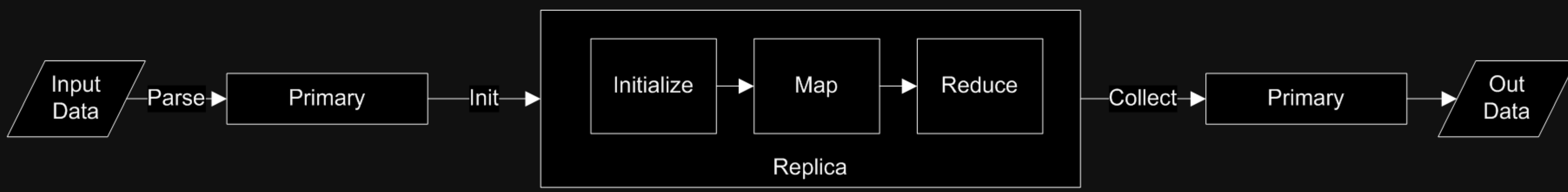
How does Repy code affect porting MapReduce functionality?

- Since methods can’t be added dynamically, map-reduce replicas must be initialized with these methods pre-processed

- MapReduce implementation in Repy is not a job manager (e.g. Hadoop), but more like an individual task manager

Primary -> Replica -> Primary

- Simple data pass, no partitioning/collecting



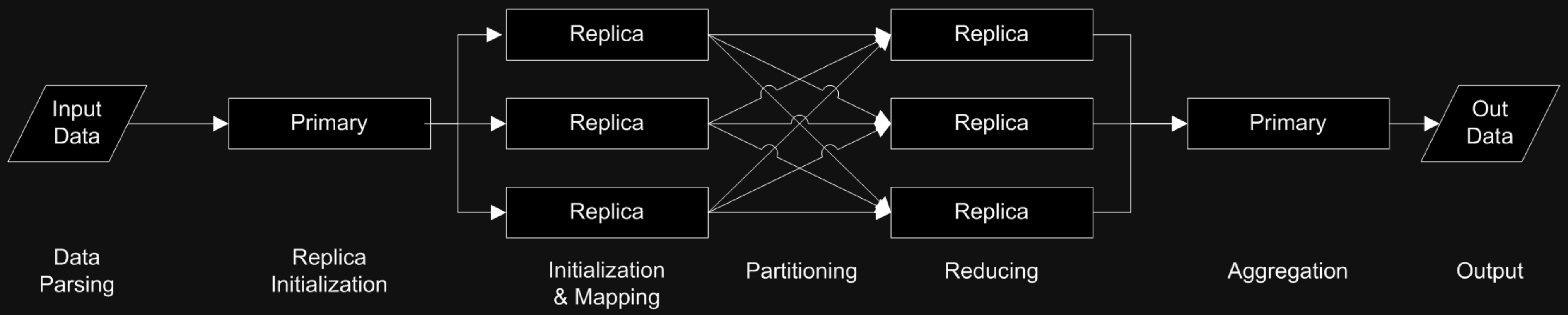
sep

- Message sending scheme: 14*128.208.1.121:

Size (B) Data

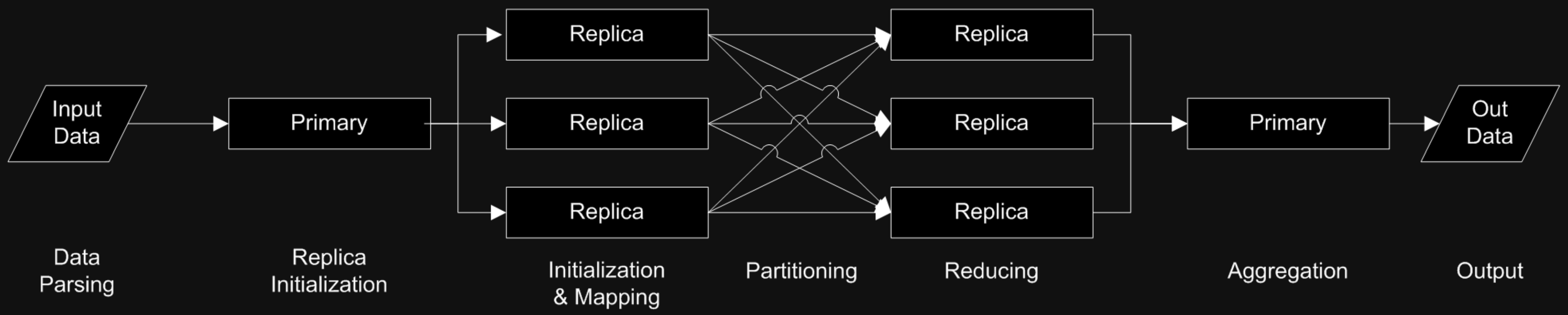
Primary -> n Replicas -> Primary

- Input data split into equal chunks for each peer
  - Another limitation of Repy (no advanced FS ops)
- What happens when a node dies?
  - Wasted work…
- Semi-transitivity of connections will halt all progress

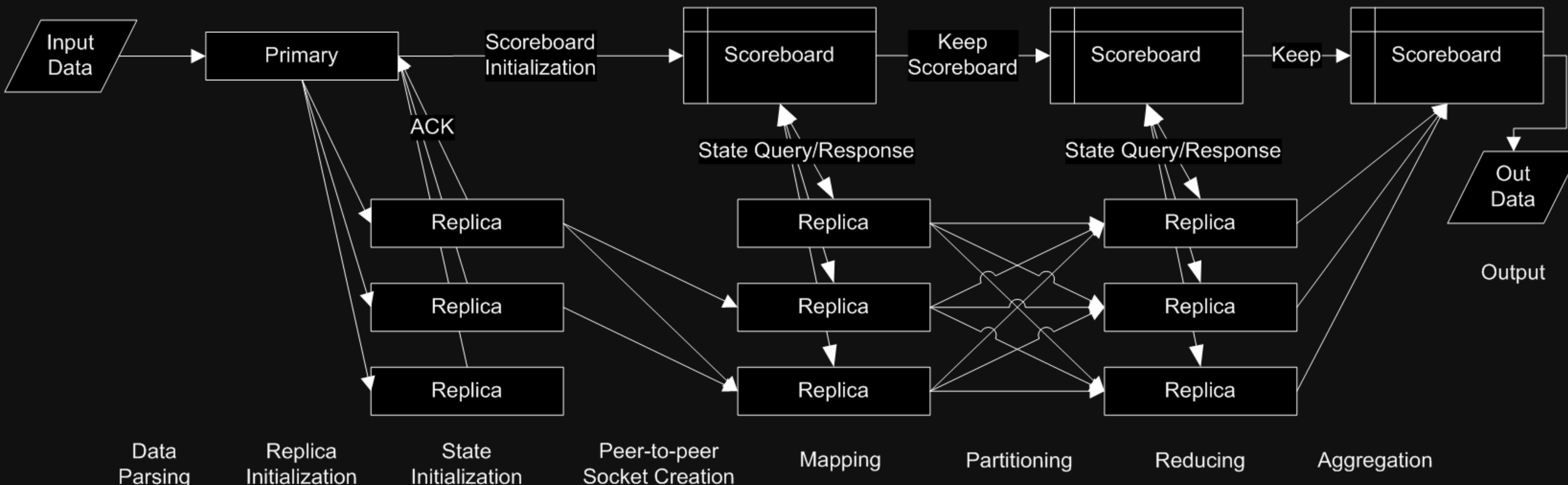


Partitioning

- A lot of python list, set, and dict mutations to arrive at a list of data to send to each node
  - List of (k,v) -> list of h1: (k1,v1,v2), h2: (k2, v3) .. -> list of n1 -> (k1, [v1,v2,v3]), n2 -> ...
- Needs to hold the property that identically hashed keys get shuffled to the same reducer.



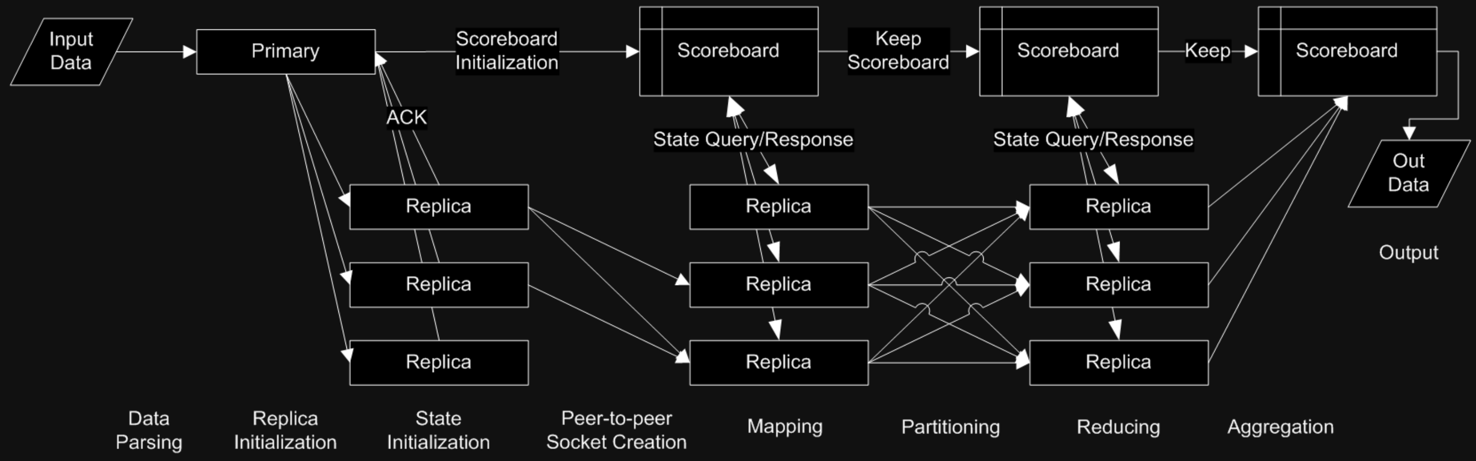
Add in some preliminary fault-tolerance..



- Primary keeps a 'scoreboard' of replicas
- ACKs implemented to ensure all nodes get initialized
- Peer-peer sockets initialized and retained early

Avenues for fault-tolerance

- Fix semi-transitivity problem by replacing active replicas with inactive ones
- Use a new Repy feature (timeout sockets) to poll for new data or to abort trying after a specified timeout
- Either the primary or any node can request a new node for a downed node.
  - All the primary needs to know is the index of the old node



Placing MapReduce on Seattle

- Since Seattle vessels are distributed across the world, many issues arise:
  - Method of selecting ideal node for primary
    - Selection process by central location, proximity to user's location, lowest ping, lowest avg hop route?
  - Variable latency issues
  - Semi-transitivity between all nodes + primary
  - Bandwidth issues

Demo?

- Three nodes on LAN
  - One primary parses, distributes, scoreboards and aggregates
  - Two nodes map, partition, and reduce data
- Simple word-count example!

Future Work

- Clean up and refactor code
  - This is an early use of Seattle for computational means; it should be a model for new developers!
- Add additional fault-tolerance capabilities, test extensively on Seattle
- Add user-interface – a Seattle node can easily become a webserver (in 6 lines of Repy!)

Acknowledgements

- Ivan Beschastnikh (UW) - debugging prowess, protocol planning help, and gentle prodding to work faster!
- Charlie Garrett (Google) - valuable discussions on implementing fault tolerance and detailing strategies that Google employs to partition, read, and shard data
- Aaron Kimball, Slava Chernyak, and Ed Lazowska (UW) - introducing us all to the wonders of map-reduce and exposing us to such wonderful and influential engineers and managers working in the cloud
- Justin Cappos (UW) – The initial idea and vision for Seattle

alpers@cs.washington.edu